Common Flaws of Distributed Identity and Authentication Systems

Brad Hill, brad@isecpartners.com

February 2011

Author's Abstract

This paper presents an informal list and plain-language discussion, in the spirit of the "OWASP Top 10", of some common flaws in distributed authentication, authorization and identity systems of the last fifteen years. Those inventing, implementing, deploying and evaluating such systems may find the list useful in avoiding similar mistakes. Examples from the literature and author's personal experience are discussed.

Introduction

To make access to diverse and distributed information resources easier and more secure for users, distributed authentication systems are now a part of almost every major information technology system, enabling single-sign on, identity federation, delegation, mash-ups and more. Widespread adoption of these systems, even in enterprise contexts, has mostly happened only in the last decade, and the last five years have seen many new protocols and implementations targeting the Web and mobile systems. As the pace and quantity of these systems' invention and implementation has accelerated, so has the history of flaws and errors grown, but little attempt has been made to catalog recurring mistakes or anti-patterns in the last fifteen years, except in the context of much larger textbooks on cryptographic engineering. This paper attempts to provide, based on a survey of the literature and the author's personal experience examining many such systems, practical advice for recognizing and avoiding the most common weaknesses encountered in modern distributed authentication systems.

Background

Although the goal of this work is to produce a short and approachable summary, the following papers are recommended for all designers and implementers of cryptographic protocols, and require little to no background in the mathematical formalisms of cryptography. There are many excellent papers on the topic of distributed authentication; these are merely some of this author's favorites.

Prudent Engineering Practice for Cryptographic Protocols, Abadi and Needham, 1995
Robustness Principles for Public Key Protocols, Anderson and Needham, 1995
Programming Satan's Computer, Anderson and Needham, 1995
Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure, Ellison and Schneier, 2000
Authentication in Distributed Systems: Theory and Practice, Lampson, Abadi, Burrows and Wobbler, 1992
Ceremony Design and Analysis, Ellison, 2008

For those who have a higher tolerance for notation, the following are also recommended:

Using encryption for authentication in large networks of computers, Needham and Schroeder, 1978
Trust Relationships in Secure Systems – A Distributed Authentication Perspective, Yahalom, Klein and Beth, 1993
A taxonomy of Replay Attacks, Syverson, 1994
Some New Attacks upon Security Protocols, Lowe, 1996
Federated Identity-Management Protocols (Transcript of Discussion), Pfitzmann, 2005

Excellent books providing a broad background on the subjects of cryptography, protocol engineering and authentication include:

Cryptography Engineering: Design Principles and Practical Applications, Ferguson, Schneier and Kohno, 2010
Security Engineering: A Guide to Building Dependable Distributed Systems, Anderson, 2008
Network Security: Private Communication in a Public World (2nd Edition), Kaufman, Perlman and Speciner, 2002
.

The common flaws:

Unconstrained Delegation

A credential can be delegated if, when you give it to somebody, they can use it not just to authenticate you, but to authenticate *as* you, to somebody else. This is a useful property, but a dangerous one, especially when it is not an explicit requirement of the system.

The username and password are the best example of a delegable credential. That passwords suffer from unconstrained delegation, not only within, but across system boundaries, has spawned an entire industry of fraud. Some ability to delegate authority is a highly desirable property but unconstrained delegation places too much trust in recipients of credentials and amplifies the consequences of any errors.

In attempting to replace passwords with other security tokens, often only some of the undesirable delegation properties are addressed. Designers of systems should ask, if a user would be unwilling to give their password to a party who wants to act on their behalf:

- Why are they unwilling?
- Does my system constrain delegation in a way that addresses these concerns?

Typical reasons a user might be uncomfortable with delegating their password might include:

- Inability to audit or attribute actions taken on behalf of a user to the delegated-to party.
- Inability to grant a limited subset of user rights and privileges.
- Inability to grant access for a limited time or limited number of actions.
- Inability to revoke access.

A good protocol will address all of these concerns, not just provide a password-equivalent by a different name.

Bearer tokens are most commonly exploited by attackers for their delegation properties. A bearer token is one which, like their namesake bearer bonds, requires nothing other than the instrument itself to be used. Bearer tokens are common in many systems and flavors: SAML, OAuth2, HTTP cookies, even Kerberos in some usages. Bearer tokens are attractive targets to attackers because there are many ways in which they may be disclosed in complex systems – cross-site scripting, SQL injection, confused deputy attacks or simple information disclosure flaws.

That bearer tokens may have an expiration date and can be revoked allow the customer experience to be better in the event of a data breach, but typically only after the damage of an attack has been done. Consider that credit cards numbers are also bearer tokens – their expiration period and ability to be revoked has not made them any less interesting to criminals.

Bearer tokens may be necessary in some situations, such as redirect-based protocols with passive requestors (browsers), but for active clients such as web services or rich mobile apps, there is rarely a good reason not to perform key agreement and require holder-of-key proof to use a token. There are well-established and standard mechanisms for doing so and the computational cost is quite low for even the least-expensive modern hardware.

Solutions

Best practices to reduce the risks of unconstrained delegation include:

- Mark authentication artifact with their intended target, as with a SAML AudienceRestriction or WS-* AppliesTo header.
- Indicate in the artifact, or in state associated with the artifact, the subset of resources or privileges authorized, rather than simply an identity.
- Avoid bearer tokens. Build key exchange / agreement into the protocol and require proof-of-possession of the key to use an artifact, to reduce risks of disclosure in transit or from data at rest.
- Limit the lifetime of authentication artifacts.
- Indicate in the artifact and in application logs the principal being delegated to, and acting on behalf of, the ultimate authorizing principal.

Unbound Composition of Transport and Message Security

A common pattern for modern cryptographic protocols is to attempt to compose the necessary properties of an authenticated exchange using both transport and message-level security. The pattern is expressed most clearly in web service security bindings referred to as “mixed mode” or “message credential with transport security”. A transport protocol, usually TLS, is used to provide confidentiality, integrity, and to authenticate the server to the client. Authentication of the client by the server is provided by an inner mechanism tunneled over the secure transport. This provides a great deal of convenience, flexibility and performance. TLS is broadly available, and the inner context token can allow

interoperability with a wide variety of existing credential types such as Kerberos, X.509 certificates, SAML tokens or username/password. For many of these token types, some level of proof-of-possession of the credential may be supplied by signing a header or timestamp, but not the entire message.

In addition to providing confidentiality and authentication, cryptography can serve to bind together parts of a message in a protocol. When message parts are protected independently, or when the protections use entirely different keys, an attacker may be able to separate the message parts and recombine them in unintended ways.

Practical exploitation is possible when an adversary can take a received authentication artifact and forward it over a newly constructed secure transport tunnel to a different endpoint. This is relatively easy for artifacts and protocols that are not scoped to a particular target server. The adversary convinces a client to connect to it and send it a credential. The adversary then connects to another server, over another TLS channel, and forwards the credential. With a bit more sophistication, even artifacts scoped to a target server may be vulnerable to forwarding. Protocols and security bindings vulnerable to this style of attack include:

- NTLM and Kerberos over HTTP(S)
- Various authentication methods of SASL and EAP
- Authentication exchanges which rely on IPSec for confidentiality and integrity
- Renegotiation in SSL and TLS for client certificate authentication
- WS-* message credentials unbound with transport security
    - Certificate tokens used without a signed and verified AppliesTo header
    - SAML tokens used without an AudienceRestriction
    - Kerberos tokens where the server SPN is specified by an un-trusted WSDL

The WS-* Kerberos case is particularly demonstrative. Kerberos tokens appear to follow many best practices. They are scoped to a particular server, negotiate key material, and require proof-of-possession by the client to use. But attacks are still possible:

Alice, our client, is about to engage in a transaction with Mallory, an (unbeknownst to Alice) malicious server. Alice retrieves a WSDL document from Mallory. Mallory's WSDL says she willing to accept Kerberos authentication, and that her Service Principal Name (SPN) is Bob. With no way to verify if this is really Mallory's SPN, Alice gets a Kerberos ticket for Bob, and forms a message. She includes as her message credential a Kerberos AP-REQ and a signature over a timestamp header using the Kerberos ticket session key. She sends this with her message to Mallory, over TLS. Receiving this, Mallory establishes her own TLS connection to the real Bob, and sends her own payload using Alice's Kerberos token and signed timestamp.

Other attacks are possible if the same token used without transport security can be re-used in a mixed mode binding. For example, if a Kerberos token intended for an endpoint which enforces a proof-of-key

policy can be intercepted and replayed against a mixed-mode binding without knowledge of the token secret.

Solutions

Channel bindings and service bindings provide solutions to this problem.

A service binding is an indication of either the intended target of authentication (when it can be determined securely), or the service endpoint name to which a credential was sent (when that can be determined securely) which is cryptographically bound to the credential. The AppliesTo header used with the WS-* X.509 Certificate Token profile is an example of a service binding, as is the AudienceRestriction of a SAML token. To prevent the first Kerberos attack described above, the endpoint name could be used as a service binding. Alice has no way to verify Mallory's real SPN, but if she included in her signature a header containing the verified TLS subject DNS name to which her message was being sent (mallory.example.com) or the TLS certificate thumbprint, the real Bob could verify if this identifies the endpoint on which he received the message.

A service binding would not stop the second attack, however, unless it was set to an explicit "null" value for tokens sent over an insecure channel.

A channel binding, in the words of RFC 5056, "allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer". To continue our Kerberos example, if Alice included in her token or signed data the TLS Finished message of the channel over which she sent her credential, or an explicit null for messages not sent over a secure channel, Bob could use this to verify that the holder of the credential was connected to the same TLS channel Bob received it over, eliminating the possibility of a man-in-the-middle.

Real-world implementations of solutions to this kind of forwarding attack include last year's "Enhanced Protection for Integrated Windows Authentication", which supplies both channel and service bindings as part of NTLMv2 and Kerberos protocol messages, and "draft-ietf-tls-renegotiation-01", which uses the TLS Finished message to bind inner cipher suite and client certificate renegotiations to the originally established outer TLS channel.

Another solution is to simply avoid "mixed-mode" security bindings. Channel binding is accomplished implicitly when the same mechanism is used to implement and bind all security properties of an exchange, such as using GSS-API with Kerberos for authentication and the negotiated session key for integrity and confidentiality.

Un-Scoped or Over-Scoped Authority

The typical identity, authentication or authorization system has some domain of names that can identify principals in the system. The problem of un-scoped or over-scoped authority arises when a party issuing claims or assertions is trusted to issue them for parts of the namespace for which it is not authoritative for, such as names outside of its bailiwick, for well-known identifiers that should only have internal meaning, or when a system does not allow the possibility of namespace partitioning.

PKIX, the global PKI used for SSL/TLS, is the worst offender in this category. Although it is possible to specify a name constraint that limits the subject names a given Certification Authority may certify, this feature is rarely implemented and even less often used. In practice, every public CA configured for a platform is trusted to certify every name. There are dozens of public authorities, and all are equally trusted, though all are not equally trustworthy. Worse yet, these same CAs are trusted by default to certify non-unique or internal names, such as single label hostnames (e.g. "mail"), non-assigned-FQDNs ("web.internalonly"), or IP addresses, effectively rendering useless any local or enterprise authority that might actually be the trusted authority for a given relying party.

This problem is also somewhat common in enterprise federation scenarios. Administrators configure a "trust relationship" with a business partner to be able to accept credential assertions or claims from them and authorize access to resources. All too often, such trust is over-provisioned. For example, the Air Force may want to accept claims from Boeing for access to some resources. However, it should take care that Boeing is only allowed to present claims for *worker@boeing.com*, and not for *administrator@airforce.mil*, or *worker@lockheed.com*. Even when systems get these restrictions correct for user principals, they often fail to do so for server principals. For federated identity protocols that don't use TLS, allowing a client to properly authenticate a remote server can be critically important. When configured for federation by the Air Force, can Boeing assert the identity of "fileserver.airforce.mil", or just "fileserver"? For protocols that do use TLS but must access resources certified by foreign enterprise authorities, the same issues previously discussed for PKIX re-emerge. There is no way for the Air Force to trust Boeing's enterprise authority to assert https://files.boeing.com/ without it also being able to assert https://internal.airforce.mil/ (or https://www.microsoft.com, for that matter) other than a mightily obscure, badly documented, inconsistently implemented and almost unused feature of X.509: cross-certification with name constraints.

Active Directory's Kerberos implementation in Windows 2000 provides two more examples of problems with un-scoped trust. The first is in the implementation of delegation. In Windows 2000, a server can be marked as "trusted for delegation". This allows the server to delegate appropriately-issued Kerberos credentials it receives to any other principal in the domain. This means that any server so marked has the same effective privileges as the Domain Controller: so long as it can lure a given user to authenticate to it, it can use that user's credentials anywhere. As a result, most security guidance recommended against ever using this feature. In Windows Server 2003, Kerberos Constrained Delegation (A2D2) was introduced. Servers configured for A2D2 are only allowed to delegate to certain, administratively designated, principals.

The other example of unconstrained authority in Windows 2000 Kerberos came from cross-forest trusts. Using the SID history feature, the Privilege Attribute Certificate included with Kerberos tickets from a foreign realm could include SIDs for identities in the local forest, including well-known, privileged identities, allowing an elevation of privilege by the foreign KDC. To address this issue, in 2002, Microsoft added SID filtering and quarantine, which removes any SIDS that are not relative to the trusted domain from any authorization data that is received from that domain.

Solutions

Define the bailiwick or scope of trust for an authority as an inherent part of protocols, when possible. Make configuration of this a mandatory part of establishing a connection between two systems and filter any claims not relative to the issuing authority.

Though it is simply a matter of language, the author prefers to use the phrase "configure a federation counterparty" to the more common "establish a trust relationship" when describing cross-domain trust establishment ceremonies to highlight the partially cooperative, partially adversarial nature of any such relationship and discourage over-broad grants of authority.

Avoid the use of unqualified or "local" names and identifiers in distributed systems. The days of closed networks are over.

PKI, PKIX and SSL/TLS Dependencies

SSL and TLS are the most successful security protocols in history, and are the only reasonable option for securing the HTTP traffic that constitutes a huge segment of Internet traffic today. Although many consider PKIX to be a failure as a distributed identity technology due to the lack of uptake of client certificates outside of the enterprise, it remains the most successful global-scale, universally adopted and interoperable distributed identity system. As such, even directly competing systems and protocols have strong incentives to use, interoperate with and build upon SSL/TLS and PKIX.

This is not, in itself, a flaw, but incorrect assumptions about how PKIX works can quickly lead to mistakes. A few things that should be kept in mind by protocol designers hoping to bootstrap from or interoperate with PKIX include:

- Public CAs are trusted by most client libraries to certify, and have a history of actually issuing credentials for, non-unique names. Examples of such names include unqualified DNS names (e.g. "mail"), IP addresses, and names that do not end in a recognized TLD (e.g. "secure.instantssl").
- A client certificate from a public CA has little or no assurance value. They can be obtained for free and are verified only by unprotected email messages. Only the E component of the Subject DN has any meaning at all. The CNAME component of the Subject DN of a client certificate from a public CA might contain literally anything.
- Some public CAs have been willing to issue certificates with any OID extension not in the set they explicitly recognize and distinguish on, which might be as small as [Basic Constraints, Subject, EKU].
- Many PKIX client libraries do not verify CRLs or OSCP information. .

TLS has also been subject to a variety of unfortunate breaks in the last five years. Protocol authors may argue, especially for protocols specifically targeting the World Wide Web, that their work can never be more secure than HTTPS. If HTTPS is broken, security for the entire Web is broken. This is true to a point, but authors should be cautious about putting all of their eggs in the HTTPS basket, with all of its associated complexity. At the very least, authors of high-value identity systems should be cautious that

a short window of vulnerability in TLS does not expose their users to indefinite risks such as the exposure of long-lived bearer tokens.

Distributed authentication systems that depend entirely on TLS for security are also often poorly configured. Many organizations do not use valid certificates for testing due to the costs of obtaining such a certificate from a commercial CA or the effort involved in configuring a test CA. Two common failures arise as a consequence: Either the valid certificate credentials used in production environments are also used in development, test and staging environments, breaking the principle of separation of duties and exposing this key to many unjustified parties, or test systems are configured to disable certificate checking, and these checks are never re-enabled when the system goes live.

Solutions

Avoid protocols that rely exclusively on TLS to make all of their security guarantees. Use additional mechanisms to provide redundant protection for high value messages. The discipline involved in creating protocol messages that *can* be authenticated (as opposed to, e.g. HTTP) is quite useful for designing robust protocols, even if such protections are optional and not employed in the common case.

Where HTTPS is the only protection for messages in an identity system, ensure that high value messages have a short lifetime. A momentary compromise of HTTPS should not result in a permanent compromise of individual users' identities or of the entire identity system.

Impedance Mismatch in Identity Contexts

Formal methods of analysis can provide strong proofs of the assumptions of a given protocol. Unfortunately, real distributed systems interoperate, and protocol messages and identities, claims and attributes often cross between systems with different definitions for the scope and meaning of these artifacts or concepts. Formal analyses rarely attempt to cross these boundaries and it is often at these "joints" that security problems arise.

Confusion about the granularity or scope of an identity or authN/authZ scope is a common cause of problems at interoperability boundaries. Consider the following list of the different scopes for an "identity" in some common systems:

System	Scope of "Identity"	Wildcards allowed	Implied or explicit hierarchical structure
PKIX	Any X.509 or LDAP Distinguished Name for a Subject, plus Subject Alternative Names. Distinctions on KU, EKU and other arbitrary extensions in an application dependent	Application dependent	Application dependent
TLS	The Subject CNAME as a DNS name, or DNS type Subject Alternative Name in an X.509 certificate. Single label names and IP addresses are also allowed.	For initial component only, and only below TLDs	No
DNS	A fully-qualified Domain Name, zone key. ….	Yes	Yes
Active, browser-rendered content	Same-Origin Policy: host, port, protocol, but not path	Yes	Yes – a script can re-set self.domain to parent domain
Active, browser-plugin content	Variants on Same-Origin Policy, typically adding site of code origin	Application dependent	Application dependent
HTTP WWW-Authenticate header	Site, for Basic, Negotiate, NTLM, Kerberos. URL sub-path for Digest	No	No
HTTP cookies	Same-Origin Policy, plus URL path	Yes	Yes
Kerberos	SPN or UPN	No	No
NTLM	SPN/UPN, but effectively Active Directory Forest	No	No
OAuth	Opaque identifier	Application dependent	No
OpenID	Email address or URI	No	Yes
Strict Transport Security header	Host	No	No
Web server	Host, or URL path, e.g. for blogs	N/A	Full host implied as administrative scope.
Web service	Host, port and protocol, or URL path. Application-dependent.	N/A	No

Consider the difficulties this matrix presents for systems that attempt make identities and authentication artifacts portable and interoperable across protocols, or that bootstrap one form of credential from another.

How does one authenticate with Kerberos when services at https://foo.example.com/users/bob and https://foo.example.com/users/alice are different principals? How can an SPN be formed for an HTTPS service that is registered only as "*.example.com" in DNS and X.509? There is no way to reliably cover all cases in current implementations.

What if a blog or SharePoint site with a sub-hostname identity requires a user to add it to the Trusted Sites zone in Internet Explorer, or enable WWW-Authenticate header authentication? The user cannot avoid granting privileges to other identities on the same server.

Other related and subtle distinctions can open systems to attack and failure. What are the valid characters for a hostname, login name, UPN, OpenID, certificate subject, or SAML subject? Can they all be parsed safely along the full multi-protocol path and in all contexts of a composite system?

Solutions

There are no easy solutions to this problem. Avoiding errors from impedance mismatches requires careful human attention to the exact semantics and scope of the identities, authentication and authorization contexts across all the protocols and systems being transited. A few rules of thumb include:

- Specify clear rules for the meaning and scope of names in all contexts at interoperation points between protocols. It may be necessary to introduce and verify new restrictions on the scope or format of existing protocols and credentials.
- If an identity may occur in a web browser security context, avoid making it more granular than a hostname.
- Systems which specify only end-entity principals within a partitioned namespace are easier to interoperate with than those which allow hierarchies and wildcards.
-

False Dilemmas in Adoption vs. Assurance: Is Crypto "Too Hard"?

Where academics are interested in the provable security properties of a system, engineers and businesspeople are interested in adoption. Cryptosystems are designed and deployed in the real world primarily to enable commerce. An unused system provides no security benefit to anyone. A good system is one that is able to achieve broad adoption by being not just easy to use for end-users, but easy to implement and interoperate for partners.

Many of the weaknesses in recent protocols arise because well-understood security mechanisms have simply been left out. This is due to a belief that any cryptographic security measures beyond the use of TLS will create barriers to entry and hamper the success of the protocol ecosystem. In many cases,

these perceived choices between adoption and assurance are false. High assurance can often be provided with at little cost to entry and interoperability if a few basic principles are followed.

Bearer Tokens vs. Holder-of-Key Proof

Both the Needham-Schroeder protocol for key exchange in distributed systems and RSA public key cryptography were invented in 1978, when the Apple][was the pinnacle of personal computing and it had been only a year since the first Unix system sent a TCP packet. In the last decade, enterprise Kerberos and global-scale PKI have deployments numbering in the billions, and there is more software code on the typical smartphone in a teenager's pocket than may have been written in history up to 1978. We, the engineering community, should be collectively ashamed that authentication and authorization systems designed in 2010 and intended for use by protocol-aware software agents are defaulting to bearer tokens because having to "find, install and configure libraries" for basic cryptography is considered too hard, compared to "the convenience and ease offered by simply using passwords". Using a library to perform an HMAC or signature is not too difficult nor too much to expect of developers in the 21st century.

The main legitimate reason to use a bearer token is when authentication logic is being tunneled through other protocols where some of the participants are not aware of the inner protocol. Good examples include most Web single sign on schemes, such as SAML, Liberty or WS-Federation Passive Requester Profile, where a protocol-unaware web browser is used to ferry an authentication token via means of HTTP redirects. While we have been stuck with web browsers as "the only client that matters" for a decade now, the appearance smartphones and "apps" as a major consumer and emerging business platform gives us an opportunity to leave these bad habits behind.

Solutions

When protocol-aware software agents are being used, there is little reason to not create session keys and require proof of possession of such a key for important protocol messages. Problems of scale, performance and key distribution can and have been solved. Widely deployed protocols such as Kerberos , NTLM and the Liberty Alliance family of protocols provide example patterns for distributed exchange of temporary session keys and authorization material. They have proven both fast and scalable in extremely large and distributed deployments.

When bearer tokens must be used, if a key is known for intended recipients, encryption to make relevant portions of a bearer token confidential to the intended recipient can offer additional protection. (Although receivers should be careful not to confuse confidentiality with authentication in such cases.)

For use cases that genuinely require non-software actors be able to participate in a full protocol exchange (e.g. humans with no more than cut-and-paste as a toolkit), see the "Build Two Protocols and

Incentivize" discussion below, or write a tool dedicated to the particular human interaction, such as a PowerShell commandlet or the GoogleCL[1].

Canonicalization and Transformation

In the author's experience, the perception that proof-of-possession mechanisms for tokens are "too hard" often arises from a particular set of confusions and design choices. Performing basic encryption operations is not hard; canonicalizing messages to be signed can be. Canonicalizing XML turns out to be *very* hard. Many of the difficulties around interoperable signing and verification in OAuth 1.0 were related to canonicalization. Though counterintuitive, experience seems to indicate that it is easier to write a lenient parser than a strict serializer.

The idea expressed in XML Digital Signatures v1.0 that one should "sign what is seen", not what is said, turned out to be a misleading intuition, and a layering violation. Cryptography can only deliver a few things: it can provide confidentiality, guarantee authenticity, bind together the parts of a message, and serve in producing random numbers. Ensuring that every well-formed message has an unambiguous meaning MUST be the responsibility of higher layers of a protocol. Introducing complex canonicalization or transformation mechanisms (such as XSLT) directly into the protocol operations intended to guarantee authenticity or bind parts of a message together makes the implementation of such systems difficult and prone to security errors.

Solutions

Protocol designers should strive to avoid designs where:

- Active intermediaries rely on being able to transform the literal contents of a message in a signature-preserving manner that preserves cryptographic guarantees authenticity, integrity or binding.
- A sender of a message can request complex operations of a receiver as part of verifying the authenticity, integrity, or scope of a message.

If these are absolutely necessary features, a protocol should also provide a "point-to-point literal" mode with minimal requirements for message canonicalization and transformation. The total computing resources necessary to authenticate a message should be deterministic and depend only on the message size, key size and cryptographic algorithm.

Key Discovery and Revocation Checking

The same complexity concerns around canonicalization and transformation during message processing exist for key discovery and revocation checking, but are more difficult to avoid. Since these are often network operations, they violate the previously stated principles that authenticating a message should require a deterministic set of resources, and that message senders should not be able to compel behavior of message receivers.

---

[1] http://code.google.com/p/googlecl/

Solutions

Operations required to authenticate a message, including checking the revocation status of a credential or retrieving a public key, should never require client-authenticated operations to avoid the possibility of becoming a confused deputy. Examples of bad and good requirements for verifying a protocol message include:

> **Bad:** My key is the result of the following XSLT transformation of the resource at: [*CIFS UNC path*]
>
> **Good:** My key is the entire content of the resource at: [*anonymously accessible HTTPS URL or LDAP DN*]

Processing the content of a message may require more complex operations, but these should only be undertaken once the message itself can be authenticated and, therefore, those actions authorized.

Where key retrieval must be authenticated, (such as when retrieving a symmetric key) protocols should specify and receivers expect a symbolic name, rather than an arbitrary URI. Use of symbolic names forces protocol participants to define policies around how to locate and authenticate resources. Use of arbitrary URIs for key retrieval or revocation checking often results in implementations delegating these tasks blindly to platform or library facilities with much greater functionality than necessary, creating significant risks of confused deputy or denial-of-service attacks.

Avoid Options. Build Two Protocols and Incentivize.

Ian Grigg of financialcryptography.com offers as his third hypothesis of secure protocol design: “There is one mode and it is secure." Though an admirable sentiment, the experience of the author is that a more realistic summary of real world systems is: “Something’s gotta give.” If a protocol has only one mode, its security will be reduced until any perceived barriers to adoption are removed.

In the “Introduction to Cryptographic Protocols” chapter of their book *Cryptography Engineering*, Ferguson, Schneier and Kohno give a section of discussion devoted to incentives, calling it “a fundamental component of any analysis of a protocol.”[2]

Where conflicts exist in the incentives for design of a protocol around adoption versus assurance, they may sometimes be best resolved by creating two protocols: one with low assurance and a low barrier to entry, and one with higher assurance and somewhat more implementation complexity.

This is not the same as creating different options for a protocol. Cryptographic algorithm agility and quality-of-service options can create different assurance levels – but at the cost of making a protocol still more complex and error prone. Having logically distinct protocols helps avoid imposing the unwanted costs (adoption vs. risk) of each class of user on the other.

---

[2] Ferguson, Schneier and Kohno, *Cryptography Engineering,* Wiley, 2010, Page 215

The different protocols should have a built-in incentive structure to encourage self-sorting of users. For example, the low-assurance protocol might have a different fee structure associated with its use than the high assurance protocol. An example of this type of system is Google Checkout. Both signed and unsigned shopping cards can be used by merchants, with a lower fee schedule for signed carts. The unsigned cart is easy to implement and provides a low barrier to entry that allows Google to expand its reach in the market, but high value customers have an inherent economic incentive to invest in implementing the more secure protocol to take advantage of lower fees.

Where fee structures don't apply, functionality of the low-assurance protocol might be a subset of the high-assurance protocol. The SXIP protocol family is an example of this. SXIP provides an unsigned, name value pair option to allow easy adoption for consumers of low-value identity claims, but higher-value claims may be available only with the signed XML version of the protocol.

Advantages of having two distinct protocols with a proper incentive structure include:

- Distinct protocols reduce the implementation complexity for both the high and low assurance participants compared to one protocol with multiple options.
- Distinct protocols make clear which one is in use and what security guarantees are provided, reducing errors arising from improper configuration or defaults.
- If a low assurance protocol becomes the target of active fraud and attack, it can be disabled with no impact to customers already using the high assurance version. Those using the vulnerable protocol can be clearly identified and have a secure, tested, and already operational upgrade path available in the high assurance protocol.

Implementation Foibles

Beyond the design flaws so far discussed, the implementation of even a correct protocol presents many possibilities for errors by the inexperienced. Among the most common observed by the author include:

- Failing to verify signatures on security tokens. This is especially common for tokens which are encrypted. Developers without a good understanding of public key cryptography incorrectly assume that receipt of a message encrypted to their key can guarantee authenticity of the sender, or that a public key can be kept private. Necessarily though, anyone with knowledge of a public key can send such a message, and the one-way nature of the problems public key cryptography is built on often means that public keys cannot be kept private from an adversary who can see a number of encrypted messages.
- Timing side channels in HMAC verification. Standard byte array comparison algorithms can leak timing information that may allow brute force guessing to be performed incrementally, and in linear time relative to key size. This common bug can be foiled by constant-time array comparison methods, or by randomization through re-applying the same HMAC algorithm to the received and calculated HMAC values prior to comparing them.
- Use of insufficiently random identifiers. Many protocols call for the use of a Universally or Globally Unique Identifier (UUID or GUID) as an artifact identifier, token reference, session ID, or other form of security capability. Often, the security of these protocols depends critically on

these identifiers remaining secret and un-guessable. The V1 UUID algorithm is designed to ensure uniqueness for persistent identifiers, but not to make these identifiers difficult to guess. Use of a cryptographically strong random number generator to produce 128 bit values should provide sufficient collision resistance, even for distributed systems, especially for identifiers with a short lifetime. The V1 algorithm should generally be avoided for protocol artifacts.

- Use of encryption to provide confidentiality without integrity / authentication. While allowing an agile choice of cryptographic algorithms is a strong point for a protocol's design, such protocols often offer little or no guidance on the proper composition of registered algorithms in order to achieve all necessary properties of the system. Integrity protection should be mandatory over all encrypted material to prevents attacks such as block re-ordering in ECB, bit flipping in CBC or stream ciphers, and padding oracles that allow complete inversion of CBC ciphers.

Solutions

After a decade of advocacy by Bruce Schneier and others, the software community has come to accept that amateur encryption algorithms, and even amateur implementations of proven algorithms, are a fundamentally bad idea. We should strongly consider that the same may be true for distributed authentication protocols.

For those participating in distributed authentication protocols, the prevalence observed by the author of these basic flaws points to the importance of:

- Choosing libraries developed and actively maintained by expert cryptographers
- Keeping those libraries up-to-date
- Engaging expert review for any internal implementations of such systems
- Creating acceptance criteria and audit requirements to verify the correctness of counterparties and partners in your distributed authentication ecosystem

Conclusion

The author has found the principles and examples described in this paper useful in his own work. As the creation and use of cryptographic protocols becomes more democratized and less formal, it is hoped that these guidelines will help improve the design, implementation, assessment, and integration of counterparties into both new and existing distributed authentication systems.